University of Pittsburgh

Object-Oriented Programming Cont.

Week 10

MEMS 1140—Introduction to Programming in Mechanical Engineering

Learning Objectives (L.O.)

At the end of this lecture, you should understand/be able to:

- ❏ What Object arrays are;
- ❏ Interacting with Object arrays;
- ❏ What Inheritance is.
- ❏ Implement Inheritance for beam bending.

Table of Contents (ToC)

1 – What are Object Arrays

We've used arrays before in this course. For example:

```
x = linspace(0, 1, 100);
```

An object array functions in the same way, but each element is an instance of a class, instead of just a number.

Consider the class from the previous lecture. A 5-element array would have 5 different objects of **`MaterialSample`**.

2 – Explicit Array Creation

Consider the following array of **`MaterialSample`** objects:

```
>> material_samples = [MaterialSample(193e9, 0.01^2, 1), ...
                       MaterialSample(193e9, 0.02^2, 1), ...
                       MaterialSample(193e9, 0.03^2, 1)]
material_samples =
  1x3 MaterialSample array with properties:
    E
    A
    L
```

Command Window

2 – Accessing Array Elements

Like normal arrays, this array can be indexed for each element:

```
>> material_samples(1)
ans =
  MaterialSample with properties:
    E: 1.9300e+11
    A: 1.0000e-04
    L: 1
```

Command Window

The first element is an object with **A** of 1 [cm^2].

2 – Accessing Array Elements

Like normal arrays, this array can be indexed for each element:

```
>> material_samples(2)
ans =
  MaterialSample with properties:
    E: 1.9300e+11
    A: 4.0000e-04
    L: 1
```

Command Window

The second element is an object with **A** of 4 [cm^2].

2 – Accessing Array Elements

Like normal arrays, this array can be indexed for each element:

```
>> material_samples(3)
ans =
  MaterialSample with properties:
    E: 1.9300e+11
    A: 9.0000e-04
    L: 1
```

Command Window

The third element is an object with **A** of 9 [cm^2].

2 – Automatically Creating Arrays

Manually populating an object array is tiring.

There are two easy ways to create an array with many instances of a class:

- Append `.empty(sz1,sz2)`;
- Initialize element 1 of the array equal to an object and then append more.

We'll demonstrate both methods.

2 – Using `empty`

Space for 100 instances of **`MaterialSample`** is initialized:

```
>> samples = MaterialSample.empty(0,100)
samples =
  0x100 MaterialSample array with properties:
    E
    A
    L
```

Command Window

Note the dimension 0. We can populate each element later.

2 – Initializing Each Empty Element

We can loop over the array to initialize each object with different properties:

```
areas = linspace(0.01^2, 0.03^2, size(samples,2));
for sample_index = 1 : size(samples,2)
  samples(sample_index) = MaterialSample(193e9, areas(sample_index), 1);
end
```

This fully populates the array of **`MaterialSample`** objects.

2 – Growing the Array

Alternatively, we can grow the array one object at a time:

```
areas = linspace(0.01^2, 0.03^2, 100);
for sample_index = 1 : size(areas,2)
  samples(sample_index) = MaterialSample(193e9, areas(sample_index), 1);
end
```

Both methods produce the ***same*** array!

2 – Arrays of Object Properties

Use the dot notation accessor for all properties in the array:

```
areas = linspace(0.01^2, 0.03^2, 3);
for sample_index = 1 : size(areas,2)
  samples(sample_index) = MaterialSample(193e9, areas(sample_index), 1);
end
[samples.A]
```

```
ans =
     1.0000e-4   4.0000e-4   9.0000e-4
```

Command Window Output

2 – Acting on the Objects

We can also automatically apply the **`axialLoading`** method to each object in the array:

```
tested_samples = MaterialSample.empty(0,100);
for sample_index = 1 : size(samples,2)
  tested_samples(sample_index) = axialLoading(samples(sample_index),3e6);
end
```

This process constitutes testing 100 different samples for easy behavior analysis.

2 – Collecting the Test Data

To graph the test results, get the new lengths and calculate ε:

```
new_sample_lengths = [tested_samples.L];
sample_strains = (new_sample_lengths - 1) / 1; % delta_L / original_L
.
.
.
.
.
.
.
.
```

2 – Collecting the Test Data

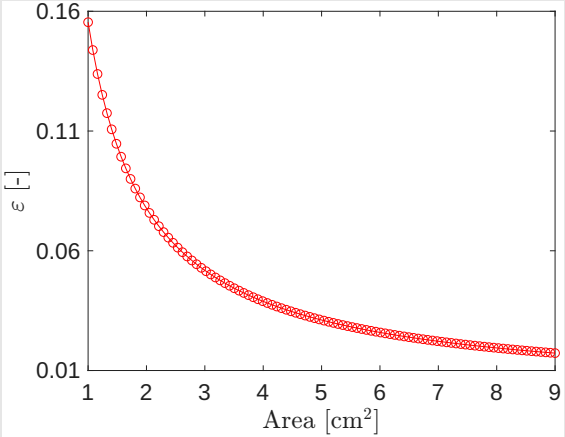
To graph the test results, get the new lengths and calculate ε:

```
new_sample_lengths = [post_test_samples.L];
sample_strains = (new_sample_lengths - 1) / 1; % delta_L / original_L
plot(areas ./ (0.01^2), sample_strains, 'o-r')
xlim([1,9])
xticks(1:9)
yticks(0.01 : 0.05 : 0.16)
ylim([0.01, 0.16])
xlabel('Area [cm$^2$]', 'Interpreter', 'latex')
ylabel('$\varepsilon$ [-]', 'Interpreter', 'latex')
set(gca, 'FontSize', 16)
```

2 – Test Results Graph

This produces the following figure.

This illustrates the inverse relationship between strain and the cross-sectional area!



3 – Inheritance

Inheritance, as a concept, enables classes to adopt certain properties and methods from another class.

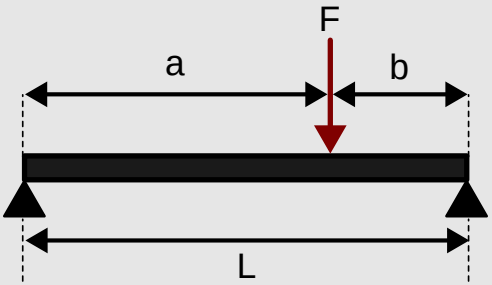
This is great when you have to create several slightly different classes that all share common properties or methods.

For example, a class **`Beam`** would have general implementation details for beam bending behavior.

But there are different beam cross-sections, so evaluating specific properties should be handled separately.

3 – Beam Bending

You'll get to this in Statics 2, but we're going to implement the deflection equation for a simply-supported beam.



$$\delta = -\frac{Fbx}{6LEI}\left(L^2 - b^2 - x^2\right)$$

$$0 \leq x \leq a$$

3 – `Beam` Deflection Generality

The beauty of this problem is that the deflection equation itself *does not vary* for any specific cross-section properties.

Only the value of the second-moment of area `I` varies, which should be handled by each subclass.

Therefore, we can implement the general deflection equation directly into the **`Beam`** superclass in our class definition.

4 – `Beam` Superclass

First, the superclass constructor is left empty because each subclass will have its own implementation for these properties.

```
classdef Beam
  properties
    E, A, L, I
  end
  methods
    function obj = Beam(~)
    end
  end
end
```

4 – `Beam` Deflection Implementation

Then we can write a general function for the earlier δ equation:

```
function deflection = beamDeflection(obj, F, a, b)
    .
    .
    .
    .
    .
    .
    .
    .
end
```

4 – `Beam` Deflection Implementation

Within it, define an anonymous function for the math itself:

```
function deflection = beamDeflection(obj, F, a, b)
  deflection_func = @(x, b) ...
    -F*b.*x ./ (6*obj.L*obj.E*obj.I) .* (obj.L^2 - b^2 - x.^2);
  .
  .
  .
  .
  .
  .
end
```

4 – `Beam` Deflection Implementation

Define an `x` array and deflection from `0` to the force application:

```
function deflection = beamDeflection(obj, F, a, b)
  deflection_func = @(x, b) ...
    -F*b.*x ./ (6*obj.L*obj.E*obj.I) .* (obj.L^2 - b^2 - x.^2);
  x_a = linspace(0, a, ceil( 100 * (a/obj.L) ));    % x from 0 to F
  y_a = deflection_func(x_a, b);                    % y from 0 to F
  .
  .
  .
  .
end
```

4 – `Beam` Deflection Implementation

Define an **`x`** array and deflection from the force application to **`L`**:

```
function deflection = beamDeflection(obj, F, a, b)
  deflection_func = @(x, b) ...
    -F*b.*x ./ (6*obj.L*obj.E*obj.I) .* (obj.L^2 - b^2 - x.^2);
  x_a = linspace(0, a, ceil( 100 * (a/obj.L) ));
  y_a = deflection_func(x_a, b);
  x_b = linspace(b, 0, floor(100 * (b/obj.L) ));    % x from F to L
  y_b = deflection_func(x_b, a);                    % y from F to L
  .
  .
end
```

4 – `Beam` Deflection Implementation

Package the `x` and `y` arrays together:

```
function deflection = beamDeflection(obj, F, a, b)
  deflection_func = @(x, b) ...
    -F*b.*x ./ (6*obj.L*obj.E*obj.I) .* (obj.L^2 - b^2 - x.^2);
  x_a = linspace(0, a, ceil( 100 * (a/obj.L) ));
  y_a = deflection_func(x_a, b);
  x_b = linspace(b, 0, floor(100 * (b/obj.L) ));
  y_b = deflection_func(x_b, a);
  deflection = [x_a, obj.L - x_b(2:end); ...          % x (0 to L)
                y_a, y_b(2:end)];                      % y (0 to L)
end
```

4 – Superclass `bendBeam` Method

Then, write a generic **`bendBeam`** method just as an added abstraction:

```
function bendBeam(obj,F,a)
  b = obj.L - a;
  deflection = obj.beamDeflection(F,a,b);
  obj.plotBeam(deflection, a);
end
```

4 – Superclass `plotBeam` Method

And also write a generic **`plotBeam`** method:

```
function plotBeam(obj, deflection, a)
  plot(deflection(1,:), deflection(2,:), '.-b');
  xline(a,'-r')
  daspect([5,1,1])
  xlim([0, obj.L])
  ylim([min(deflection(2,:)) * 1.1, max(deflection(2,:)) * 1.1])
  xlabel('$x$ [m]', 'Interpreter', 'latex')
  ylabel('$\delta$ [m]', 'Interpreter', 'latex')
  set(gca, 'FontSize', 16)
end
```

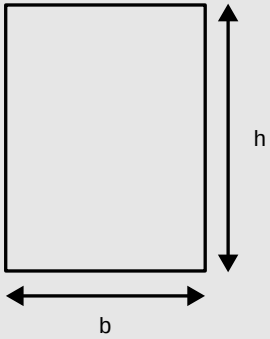
4 – Subclasses

Now let's talk about the different types of beam cross-sections.

The simple rectangular cross-section.

$$A = bh$$

$$I = bh^3$$



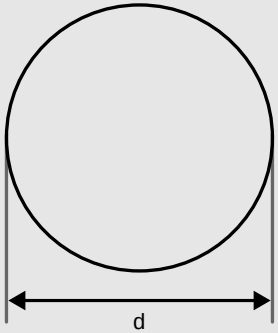
4 – Subclasses

Now let's talk about the different types of beam cross-sections.

The circle cross-section.

$$A = \pi \left(\frac{d}{2}\right)^2$$

$$I = \frac{\pi}{4} \left(\frac{d}{2}\right)^4$$



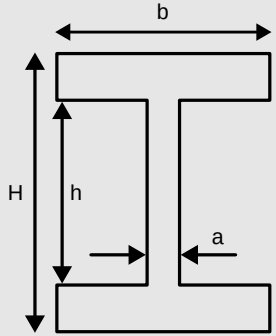
4 – Subclasses

Now let's talk about the different types of beam cross-sections.

The I cross-section.

$$A = bH - 2h\left(\frac{b}{2} - a\right)$$

$$I = \left(\frac{ah^3}{12}\right) + \left(\frac{b}{12}\right)(H^3 - h^3)$$



4 – Defining Subclasses

Each of these types of beams can be defined as a distinct subclass of the more generic **`Beam`** superclass.

For the sake of conciseness, we will only consider the subclass **`RectangleBeam`**.

We'll leave the subclasses **`CircleBeam`** and **`IBeam`** for you to do as extra practice if you are interested.

4 – `RectangleBeam` Properties

To inherit from another class, use the `<` operator:

```
classdef RectangleBeam < Beam
  properties
    b, h
  end
end
```

This enables the **`RectangleBeam`** class to use the properties that get defined as a **`Beam`** (**`E`**, **`A`**, **`L`**, **`I`**) in addition to **`b`** and **`h`**.

4 – `RectangleBeam` Constructor

The constructor accepts enough inputs to define the properties from **`Beam`** *and* the **`RectangleBeam`** subclass.

```
classdef RectangleBeam < Beam
  % <--> Properties list <-->
  methods
    function obj = RectangleBeam(E, L, b, h)
      obj.E = E; obj.L = L; obj.b = b; obj.h = h;    % User Inputs
      obj.A = b*h; obj.I = b*h^3;                    % Derived Properties
    end
  end
end
```

4 – Putting it Together

This inheritance structure enables creating a **`RectangleBeam`** and deflecting it as follows using the inherited superclass methods we wrote before:

```
youngs_modulus = 193e9;
length = 10;
base = 0.05; height = 0.1;
force_application_fraction = 0.9;
.
.
```

4 – Putting it Together

This inheritance structure enables creating a **`RectangleBeam`** and deflecting it as follows using the inherited superclass methods we wrote before:

```
youngs_modulus = 193e9;
length = 10;
base = 0.05; height = 0.1;
force_application_fraction = 0.9;
rectangle_beam = RectangleBeam(youngs_modulus, length, base, height);
.
```

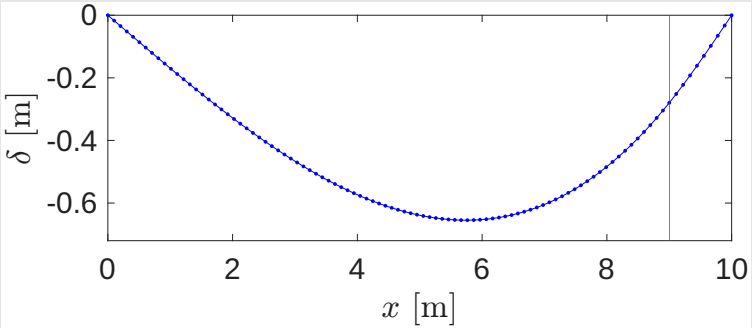
4 – Putting it Together

This inheritance structure enables creating a **`RectangleBeam`** and deflecting it as follows using the inherited superclass methods we wrote before:

```
youngs_modulus = 193e9;
length = 10;
base = 0.05; height = 0.1;
force_application_fraction = 0.9;
rectangle_beam = RectangleBeam(youngs_modulus, length, base, height);
rectangle_beam.bendBeam(1e6, force_application_fraction * length);
```

4 – Beam Deflection

This produces the following graph:



5 – Summary

This lecture covered:

- ✓ What Object arrays are

  Object arrays are a storage device for maintaining many instances of a particular class.

5 – Summary

✓ L.O.1
✓ L.O.2
✓ L.O.3
✓ L.O.4

✓ Interacting with Object arrays

They can either be created manually (tedious) or automatically by either using the `empty` method or iteratively appending another object to the array. Specific objects can be accessed by indexing through the array, just like normal. Methods and property accessors can be applied to the entire array, as well!

5 – Summary

✓ L.O.1
✓ L.O.2
✓ L.O.3
✓ L.O.4

✓ What Inheritance is

This is an approach for defining multiple classes that would all share common properties and functions, such that those commonalities are stored in a superclass, and then each subclass contains only the implementations that are not shared with other subclasses.

5 – Summary

✓ Implementing Inheritance for beam bending

Beam bending is a perfect example because the high-level behaviors of beams can be abstract in the superclass, but the specific implementation of calculating properties like area and the second-moment of area should be contained within the subclass.